Cache Consistency in Multiprocessor systems

Prof. Naga Kandasamy
ECE Department
Drexel University

May 26, 2016

The following notes were derived from:

- Culler, Singh, and Gupta, *Parallel Computer Architecture*, Morgan Kaufmann, 1999.
- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5^{th} Edition, Morgan Kaufmann, 2011.

Figure 1 shows a *symmetric (shared-memory) multiprocessor* (*SMP*) or *centralized shared-memory multiprocessor* that features a small numbers of cores, typically eight or fewer. For multiprocessors with such small processor counts, it is possible for the processors to share a single centralized memory that all processors have equal access to, hence the term symmetric. In multicore chips, the memory is effectively shared in a centralized fashion among the cores, and all existing multicores are SMPs. When more than one multicore is connected, there are separate memories for each multicore, so the memory is distributed rather than centralized. SMP architectures are also sometimes called *uniform memory access* (*UMA*) multiprocessors, arising from the fact that all processors have a uniform latency from memory.

Symmetric shared-memory machines usually support the caching of both shared and private data. Private data are used by a single processor while shared data are used by multiple processors, providing communication among the processors via reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses the data, the program behavior is identical to that in a uniprocessor. When shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. Caching of shared data, however, introduces a new problem: cache coherence.

The cache coherence problem in a multiprocessor system is both pervasive and performance critical. Figure 2 shows an example of this problem using a system with three processors whose caches are connected via a bus to shared main memory. A sequence of accesses to location u is made by the processors. First, processor P_1, reads u from main memory, bringing a copy into its local cache. Then processor P_3 reads u from main memory, bringing a copy into its cache. A little later,

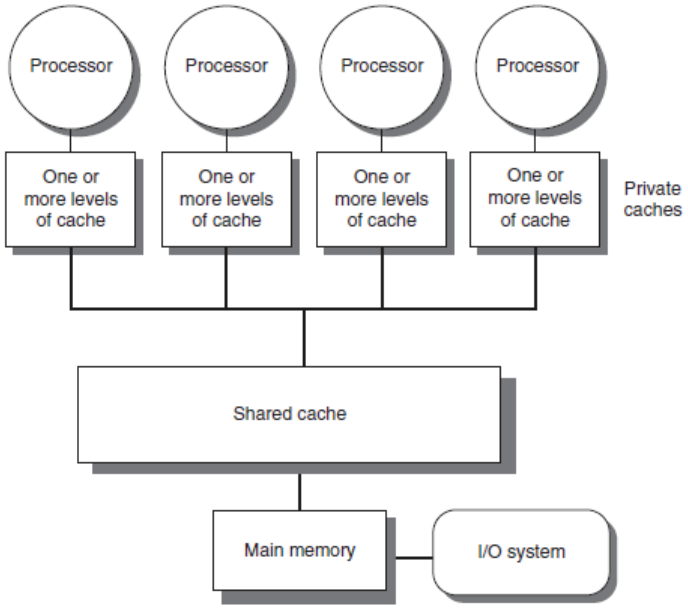

Figure 1: Basic structure of a centralized shared-memory multiprocessor based on a multicore chip.

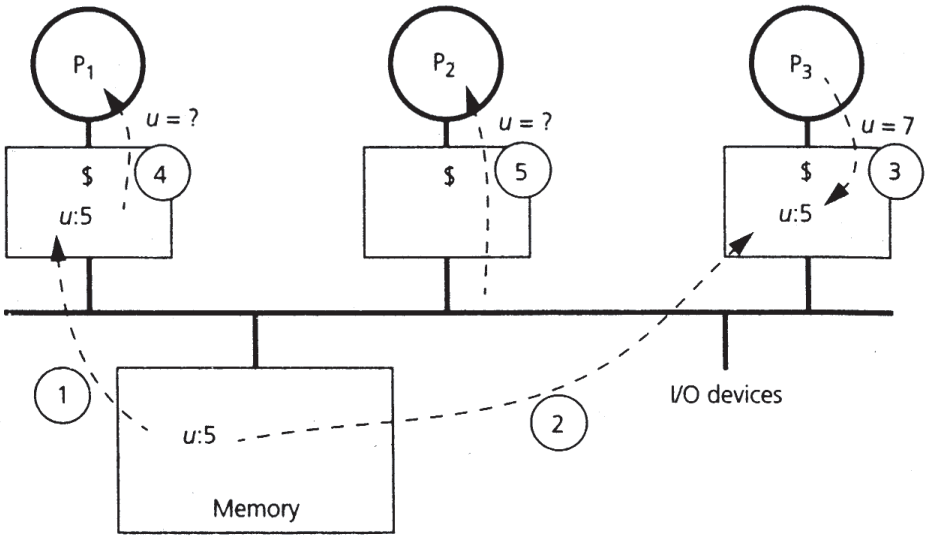

Figure 2: The cache coherence problem.

processor P_3 writes location u, changing its value from 5 to 7. With a write-through cache, this will cause the main memory location to be updated; however, when processor P_1, reads location u again (action 4 in the figure), it will unfortunately read the stale value 5 from its own cache instead of the correct value 7 from main memory.

The above situation is even worse with write-back caches. P_3's write would merely set the dirty bit associated with the cache block holding location u and would not update main memory right away. Only when this cache block is subsequently replaced from P_3's cache would its contents be written back to main memory. Thus, not only will P_1 read the stale value, but when processor P_2

reads location u (action 5), it will miss in its cache and read the stale value of 5 from main memory instead of 7. Finally, if multiple processors write distinct values to location u in their respective write-back caches, the final value that will reach main memory will be determined by the order in which the cache blocks containing u are replaced and will have nothing to do with the order in which the writes to u occur.

Formal Definition of Coherence

A multiprocessor memory system is said to be *coherent* if the following properties are satisfied:

- Operations issued by any processor occur in the order in which they were issued to the memory system by that processor.
- The value returned by each read operation is the value written by the last write to that location in the serial order.

Two properties are implicit in the definition of coherence: *write propagation* means that writes become visible to other processors (or processes); *write serialization* means that an writes to a location (from the same or different processes) are seen in the same order by all processes. For example, write serialization means that if read operations by process P_1 to a location see the value produced by write $w1$ (from P_2, say) before the value produced by write $w2$ (from P_3, say), then reads by another process P_4 (or P_2 or P_3) also should not be able to see $w2$ before $w1$. Note that there is no need for an analogous concept of read serialization since the effects of reads are not visible to any process but the one issuing the read.

Cache Coherence through Bus Snooping

We now discuss several methods of ensuring cache coherence through bus snooping. This solution to cache coherence arises from the very nature of a bus. The bus is a single set of wires connecting several devices, each of which can observe every bus transaction, for example, every read or write on the shared bus. When a processor issues a request to its cache, the cache controller examines the state of the cache and takes suitable action, which may include generating bus transactions to access memory. Coherence is maintained by having all cache controllers "snoop" on the bus and monitor the transactions, as shown in Fig. 3. A snooping cache controller may take action if a bus transaction is *relevant* to it, that is, if it involves a memory block of which it has a copy in its cache. Thus, processor P_1 may take an action, such as invalidating or updating its copy of the location, if it sees the write from P_3 to the same cache block. Since the allocation and replacement of data in caches is managed at the granularity of a cache block (usually several words long) and cache misses fetch a block of data, coherence is maintained at the granularity of a cache block as well. In other words, either an entire cache block is in valid state in the cache or none of it is. Thus, a cache block is the granularity of allocation in the cache, of data transfer between caches, and of coherence.

Fig. 3 shows how a snooping protocol maintains cache coherence. Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously snoops on the bus watching for relevant transaction and updates its state suitably to keep its local cache coherent.

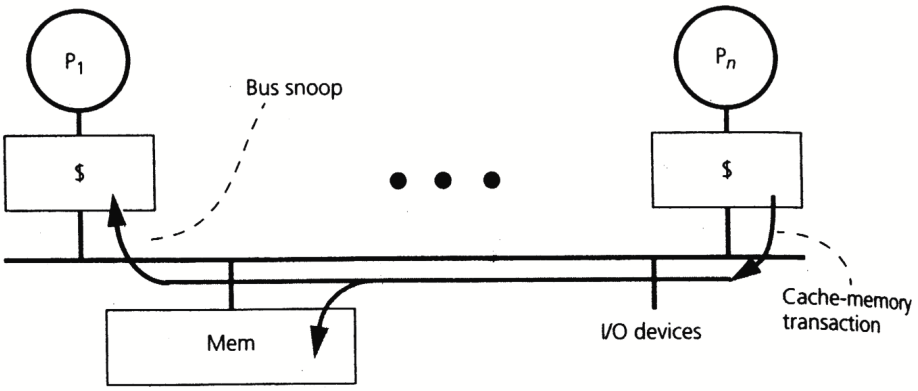

Figure 3: A snooping cache-coherent multiprocessor.

The gray arrows show the transaction being placed on the bus and accepted by main memory, as in a uniprocessor system. The black arrow indicates the snoop.

One method of enforcing cache coherence is to ensure that a processor has exclusive access to a data item before it writes that item. This style of snooping protocol is called a *write invalidate* protocol because it invalidates other copies on a write. It is by far the most common protocol. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: All other cached copies of the item are invalidated. The following table shows the cache invalidate protocol in action for two processor, A and B.

Processor activity	**Bus activity**	**Contents of A's cache**	**Contents of B's cache**	**Contents of memory location X**
				0
A reads X	Cache miss for X	0		0
B reads X	Cache miss for X	0	0	0
A writes 1 to X	Invalidation for X	1		0
B reads X	Cache miss for X	1	1	1

In the sequence shown above, we assume that neither cache initially holds X and that the value of X in memory is 0. The processor and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, processor A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it. If a multicore uses a shared cache (e.g., L3), then all memory is seen through the shared cache; L3 acts like the memory in this example, and coherency must be handled for the private L1 and L2 for each core.

Fundamentally, a snooping protocol is a distributed algorithm represented by a collection of cooperating finite state machines. It is specified by the following basic components:

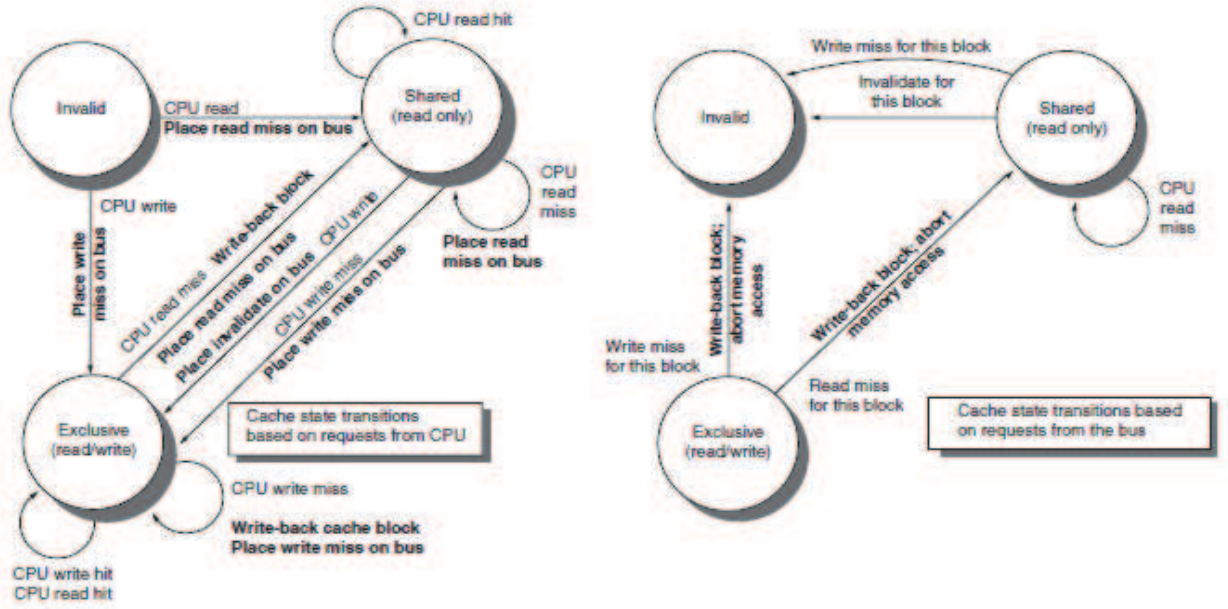

Figure 4: A write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache.

- The set of states associated with memory blocks in the local caches.
- The state transition diagram, which takes as inputs the current state and the processor request or observed bus transaction and produces as output the next state for the cache block.
- The actions associated with each state transition, which are determined in part by the set of feasible actions defined by the bus, the cache, and the processor design.

The different state machines for a block are coordinated by bus transactions.

Figure 4 shows the state machines for a simple three-state write invalidate protocol in which each cache block can be in one of three states: invalid, shared, and modified. The cache states are shown in circles. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the private cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the local cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all private caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory (or a shared cache) provides data on a read miss for a block that is clean in all local caches.

The above described protocol works due to the following reasons:

1. Any valid cache block is either in the shared state in one or more private caches or in the exclusive state in exactly one cache.
2. Any transition to the exclusive state (which is required for a processor to write to the block) requires an invalidate or write miss to be placed on the bus, causing all local caches to make the block invalid.
3. If some other local cache had the block in exclusive state when it receives a write-invalidate signal, that local cache generates a write-back, which supplies the block containing the desired address.
4. If a read miss occurs on the bus to a block in the exclusive state, the local cache with the exclusive copy changes its state to shared.

As an implementation detail, there is only one finite-state machine per cache, with stimuli coming either from the attached processor or from the bus. The state transitions in the right half of Figure 4 can be combined with those in the left half of the figure to form a single state diagram for each cache block.